CR1

// go build -race

// Sample program to show how to create race conditions in

// our programs. We don't want to do this.

package main

import (

"fmt"

"sync"

)

// counter is a variable incremented by all goroutines.

var counter int

func main() {

// Number of goroutines to use.

const grs = 2

// wg is used to manage concurrency.

var wg sync.WaitGroup

wg.Add(grs)

// Create two goroutines.

for g := 0; g < grs; g++ {

go func() {

for i := 0; i < 2; i++ {

// Capture the value of Counter.

value := counter

// Increment our local value of Counter.

value++

// Store the value back into Counter.

counter = value

}

wg.Done()

}()

}

// Wait for the goroutines to finish.

wg.Wait()

fmt.Println("Final Counter:", counter)

}

/\*

==================

WARNING: DATA RACE

Read at 0x0000011a5118 by goroutine 7:

main.main.func1()

/Users/bill/code/go/src/github.com/ardanlabs/gotraining/topics/go/concurrency/data\_race/example1/example1.go:33 +0x4e

Previous write at 0x0000011a5118 by goroutine 6:

main.main.func1()

/Users/bill/code/go/src/github.com/ardanlabs/gotraining/topics/go/concurrency/data\_race/example1/example1.go:39 +0x6d

Goroutine 7 (running) created at:

main.main()

/Users/bill/code/go/src/github.com/ardanlabs/gotraining/topics/go/concurrency/data\_race/example1/example1.go:43 +0xc3

Goroutine 6 (finished) created at:

main.main()

/Users/bill/code/go/src/github.com/ardanlabs/gotraining/topics/go/concurrency/data\_race/example1/example1.go:43 +0xc3

==================

Final Counter: 4

Found 1 data race(s)

\*/

CR2

// Sample program to show how to use the atomic package to

// provide safe access to numeric types.

package main

import (

"fmt"

"sync"

"sync/atomic"

)

// counter is a variable incremented by all goroutines.

var counter int64

func main() {

// Number of goroutines to use.

const grs = 2

// wg is used to manage concurrency.

var wg sync.WaitGroup

wg.Add(grs)

// Create two goroutines.

for g := 0; g < grs; g++ {

go func() {

for i := 0; i < 2; i++ {

atomic.AddInt64(&counter, 1)

}

wg.Done()

}()

}

// Wait for the goroutines to finish.

wg.Wait()

// Display the final value.

fmt.Println("Final Counter:", counter)

}

CR3

// Sample program to show how to use a mutex to define critical

// sections of code that need synchronous access.

package main

import (

"fmt"

"sync"

)

// counter is a variable incremented by all goroutines.

var counter int

// mutex is used to define a critical section of code.

var mutex sync.Mutex

func main() {

// Number of goroutines to use.

const grs = 2

// wg is used to manage concurrency.

var wg sync.WaitGroup

wg.Add(grs)

// Create two goroutines.

for g := 0; g < grs; g++ {

go func() {

for i := 0; i < 2; i++ {

// Only allow one goroutine through this critical section at a time.

mutex.Lock()

{

// Capture the value of counter.

value := counter

// Increment our local value of counter.

value++

// Store the value back into counter.

counter = value

}

mutex.Unlock()

// Release the lock and allow any waiting goroutine through.

}

wg.Done()

}()

}

// Wait for the goroutines to finish.

wg.Wait()

fmt.Printf("Final Counter: %d\n", counter)

}

CR4

// Sample program to show how to use a read/write mutex to define critical

// sections of code that needs synchronous access.

package main

import (

"fmt"

"math/rand"

"sync"

"sync/atomic"

"time"

)

// data is a slice that will be shared.

var data []string

// rwMutex is used to define a critical section of code.

var rwMutex sync.RWMutex

// Number of reads occurring at ay given time.

var readCount int64

// init is called prior to main.

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

// wg is used to manage concurrency.

var wg sync.WaitGroup

wg.Add(1)

// Create a writer goroutine.

go func() {

for i := 0; i < 10; i++ {

writer(i)

}

wg.Done()

}()

// Create eight reader goroutines.

for i := 0; i < 8; i++ {

go func(id int) {

for {

reader(id)

}

}(i)

}

// Wait for the write goroutine to finish.

wg.Wait()

fmt.Println("Program Complete")

}

// writer adds a new string to the slice in random intervals.

func writer(i int) {

// Only allow one goroutine to read/write to the slice at a time.

rwMutex.Lock()

{

// Capture the current read count.

// Keep this safe though we can due without this call.

rc := atomic.LoadInt64(&readCount)

// Perform some work since we have a full lock.

time.Sleep(time.Duration(rand.Intn(100)) \* time.Millisecond)

fmt.Printf("\*\*\*\*> : Performing Write : RCount[%d]\n", rc)

data = append(data, fmt.Sprintf("String: %d", i))

}

rwMutex.Unlock()

// Release the lock.

}

// reader wakes up and iterates over the data slice.

func reader(id int) {

// Any goroutine can read when no write operation is taking place.

rwMutex.RLock()

{

// Increment the read count value by 1.

rc := atomic.AddInt64(&readCount, 1)

// Perform some read work and display values.

time.Sleep(time.Duration(rand.Intn(10)) \* time.Millisecond)

fmt.Printf("%d : Performing Read : Length[%d] RCount[%d]\n", id, len(data), rc)

// Decrement the read count value by 1.

atomic.AddInt64(&readCount, -1)

}

rwMutex.RUnlock()

// Release the read lock.

}

CR5

// Sample program to show how maps are not safe for concurrent use by default.

// The runtime will detect concurrent writes and panic.

package main

import (

"fmt"

"sync"

)

// scores holds values incremented by multiple goroutines.

var scores = make(map[string]int)

func main() {

var wg sync.WaitGroup

wg.Add(2)

go func() {

for i := 0; i < 1000; i++ {

scores["A"]++

}

wg.Done()

}()

go func() {

for i := 0; i < 1000; i++ {

scores["B"]++

}

wg.Done()

}()

wg.Wait()

fmt.Println("Final scores:", scores)

}

CRa

// Sample program to show a more complicated race condition using

// an interface value. This produces a read to an inteface value after

// a partial write.

package main

import (

"fmt"

"os"

"sync"

)

// Speaker allows for speaking behavior.

type Speaker interface {

Speak() bool

}

// Ben is a person who can speak.

type Ben struct {

name string

}

// Speak allows Ben to say hello. It returns false if the method is

// called through the interface value after a partial write.

func (b \*Ben) Speak() bool {

if b.name != "Ben" {

fmt.Printf("Ben says, \"Hello my name is %s\"\n", b.name)

return false

}

return true

}

// Jerry is a person who can speak.

type Jerry struct {

name string

}

// Speak allows Jerry to say hello. It returns false if the method is

// called through the interface value after a partial write.

func (j \*Jerry) Speak() bool {

if j.name != "Jerry" {

fmt.Printf("Jerry says, \"Hello my name is %s\"\n", j.name)

return false

}

return true

}

func main() {

// Create values of type Ben and Jerry.

ben := Ben{"Ben"}

jerry := Jerry{"Jerry"}

// Assign the pointer to the Ben value to the interface value.

person := Speaker(&ben)

// Have a goroutine constantly assign the pointer of

// the Ben value to the interface and then Speak.

go func() {

for {

person = &ben

if !person.Speak() {

os.Exit(1)

}

}

}()

// Have a goroutine constantly assign the pointer of

// the Jerry value to the interface and then Speak.

go func() {

for {

person = &jerry

if !person.Speak() {

os.Exit(1)

}

}

}()

// Just hold main from returning. The data race will

// cause the program to exit.

var wg sync.WaitGroup

wg.Add(1)

wg.Wait()

}